Handling Agent Perception in Heterogeneous Distributed Systems: A Policy-Based Approach

Stephen Cranefield[1(✉)] and Surangika Ranathunga[2]

[1] Department of Information Science, University of Otago, Dunedin, New Zealand
stephen.cranefield@otago.ac.nz
[2] Department of Computer Science & Engineering, Faculty of Engineering, University of Moratuwa, Moratuwa, Sri Lanka
surangika@cse.mrt.ac.lk

Abstract. Multi-agent systems technologies have been widely investigated as a promising approach for modelling and building distributed systems. However, the benefits of agents are not restricted to systems solely comprised of agents. This paper considers how to ease the task of developing agents that perceive information from asynchronously executing external systems, especially those producing data at a high frequency. It presents a design for a *percept buffer* that, when configured with domain-specific percept metadata and application-specific *percept management policies*, provides a generic but customisable solution. Three application case studies are presented to illustrate and evaluate the approach.

1 Introduction

Multi-agent systems (MAS) technologies have been widely investigated as a promising approach for modelling and building distributed systems. In particular, much MAS research focuses on developing theories and tools that address the requirements of autonomous distributed software components that must act, interact and coordinate with each other in complex domains. Typically, agents are conceptualised as having incomplete and changing knowledge, the ability to act proactively to satisfy explicit goals, adaptive behaviour through the selection of plans that best respond to goals in a given situation, and the ability to communicate knowledge and requests to each other.

This paper considers, in particular, agents based on the popular Belief-Desire-Intention (BDI) agent model [4], which is inspired by human practical reasoning. Agent development platforms implementing this model, such as Jason [3], allow programmers to write code in terms of a dynamic belief base that is updated as *percepts* are received from the external *environment*, and *plans* are triggered by changes in beliefs and the creation of new goals by other plans. Plans can also cause *actions* to be performed in the environment. The developer must provide an environment class that models the application state visible to the agent and/or affected by its actions. At its simplest, this is a simulation of a physical environment. However, BDI agents have proven their value beyond simple simulated systems. They have been used for implementing robots [16,15], "intelligent virtual agents" [13,7,2] that control avatars in virtual worlds and

T. Holvoet and M. Viroli (Eds.): COORDINATION 2015, LNCS 9037, pp. 169–185, 2015.
DOI: 10.1007/978-3-319-19282-6_11

multi-player games, and even for real-time control of satellites [6]. As well as these situations where an agent's 'body' is controlled by software external to the agent, it may also be the case that an agent is only a component of a larger distributed system involving multiple technologies and protocols. In this case, it may be most convenient for the agent programmer to regard the external systems as part of its environment, and therefore a source of percepts and the target of actions [5].

This paper therefore considers the problem of providing an agent with a view of one or more external system components as a source of percepts, extending our previous architecture in which agent 'endpoints' act as a bridge between agents and message-based routing and mediation middleware [5][1]. There are several aspects to this problem:

1) Agents perceive the environment periodically and asynchronously from the changes occurring in the external systems. Therefore, multiple changes may occur between agent perceptions, and it is necessary to buffer these changes. **2)** BDI agents have a relatively slow execution cycle, and thus information from external systems such as virtual worlds and robot sensors may arrive much faster than the agent's perception rate. Delivering all buffered percepts to the agent on each perception may exceed its ability to trigger and execute plans. Therefore, buffered percepts should be amalgamated or summarised between perceptions. **3)** The question of whether a percept should replace an older buffered one is dependent on the domain ontology. Thus, percept buffering requires domain knowledge. **4)** The logic for summarising related buffered percepts is application-dependent. Thus, percept buffering needs application knowledge.

The first two issues above have been repeatedly encountered by researchers [6,8,12,13,10]. However, as yet, agent development tools do not provide any platform-level solution to these problems, leaving the agent programmer to implement their own application-level solutions.

This paper provides a solution to this problem, informed by the third and fourth observations above, by introducing the concepts of a *percept buffer* and configurable *percept management policies*. Together, these control the number and form of percepts provided to an agent. Given a generic percept buffer, a developer can use this in conjunction with common policies from a library or custom application-specific ones, to configure the buffer to avoid information loss and reduce the cognitive load needed for percept handling. A percept buffer therefore provides a general platform-, domain- and application-independent framework for tackling the problems of handling percepts representing information from external systems in a flexible way.

2 Related Work

The difficulty of handling high frequency percepts in BDI agent systems has been acknowledged by researchers implementing situated agents [12,13]. However, we are not aware of any implemented concrete solution to this problem.

[1] Our previous work also addresses interpreting actions as requests to external systems, but here we focus on percepts.

There is some research on abstracting the low-level sensor data received from an external environment before providing it to a BDI agent [13,16]. Similar to receiving low-level sensor data, it is also possible that the agent could receive a continuous data stream from the environment. In such a case, this continuous data stream should be discretized before providing it to the agent as percepts. Such an abstraction engine has been described by Dennis et al. [6] in the context of using BDI agents to control a satellite. By providing only abstract environment information and/or discretized information to an agent, the problem of cognitive overload can be minimised. However, this does not directly address the problem of high frequency perception—the abstracted environment information may still arrive at too high a frequency for a relatively slow BDI agent. Moreover, this previous work implements the sensor data abstraction components outside the BDI agent system, thus providing it with no control over the type and amount of the percepts it provides to agents.

An alternative approach to minimising the cognitive overload is actively filtering out percepts that do not fit certain criteria. Percept filtering is discussed alongside attention theories, where it is argued that given the fact that agent attention is a limited resource, the agent should be able to filter-out information that falls outside its current attention. Filtering can be of two-forms: top-down (goal-driven), or bottom-up. Top-down filtering refers to retaining only those percepts that are relevant to the currently pursued goals of the agent [14]. Bottom-up filtering refers to identifying salient information in the incoming percept stream that should catch the agent's attention. The work of van Oijen and Dignum [11] presents an example for goal-driven filtering of percepts by an intelligent virtual agent (IVA). When an agent adopts a new goal, it can specify the type of percepts required for that goal. This filtering is terminated as soon as the agent stops pursuing the current goal. Ideally, an IVA should be able to strike a balance between the two types of filtering.

The use of a cache or a buffer to keep environment information required by an agent is not new. For example, Oijen et al. [12] present the use of a cache to store a high level domain model derived from lower-level game state data. This information is kept until game state changes invalidate the cached derived data. Their ontology loosely corresponds to our percept metadata (see Sect. 5). However, although agents can filter the percepts they wish to perceive via subscriptions, there is no counterpart to our policies for summarising or aggregating multiple percepts received between perceptions.

3 Managing Agent Perception Using Policies

At the heart of our approach is the use of policies to manage the number of percepts produced for the deliberation process of an agent. Policies may be generic ones that are useful across a range of applications, and may be parameterised to configure them for specific applications. On the other hand, agent programmers may develop their own application-specific policies, which can be plugged into our framework via a simple interface. We also allow agents to dynamically change

the policies used to pre-process their incoming percepts in order to change the focus of their attention—an example of this is given in Sect. 9.2.

Some useful application-independent policies are listed below.

Keep latest percept. This policy will simply *replace* the previously processed matching percepts with the new one. This might be appropriate, for example, for percepts that represent sensor readings (with the sensor identifier treated as a percept *key*). If multiple readings for the same sensor arrive between two agent perceptions, the agent may only need to perceive the latest reading.

Keep latest with history. As above, this policy will ensure that at most one percept for a given functor (predicate name), arity (number of arguments) and list of key argument values is kept in the queue of percepts waiting to be perceived. However, in case the agent wishes to inspect the full recent history of matching percepts (since the previous perception), the policy records this history in the percept as an additional argument. This policy could also be refined to associate a time stamp with each percept in the history list. This policy illustrates an important feature of the design of our percept buffer: we support the use of policies that change the structure of percepts, e.g. by changing their functors and arities.

Keep most significant. Rather than keeping only the most recent percept (e.g. from a sensor), this policy will keep the one with the most significant value. For example, for a sensor monitoring Nitrogen Dioxide concentrations at a city intersection, the agent may be interested in the highest reading since the last perception.

4 Architecture

Figure 1 shows our architecture for using percept buffers to handle percept buffering, amalgamation and summarisation. We assume that percepts relevant to the agent are received via one or more channels, shown on the left hand side of the figure. These are responsible for delivering percepts obtained from external sources, such as virtual worlds, complex event detection engines and enterprise messaging systems, to the appropriate agents' percept buffers. It is the responsibility of these channels to perform whatever data preprocessing is necessary to produce percepts in an appropriate format for the agent platform used[2].

The channels also have the role of adding specific metadata to each percept to specify how the agents' percept buffers should combine this new information with any percepts that are in the buffer waiting for the agent to perceive them. Most importantly, this metadata includes the name of a *policy* to be used to amalgamate matching percepts (if required). The notion of a matching percept is defined by indicating the *key argument indices*, i.e. the argument positions that form a (possibly compound) key for a percept with that functor and arity,

[2] Eventually it may be possible to use a platform-independent format for percepts, such as the "interface intermediate language" proposed by Behrens et al. [1].

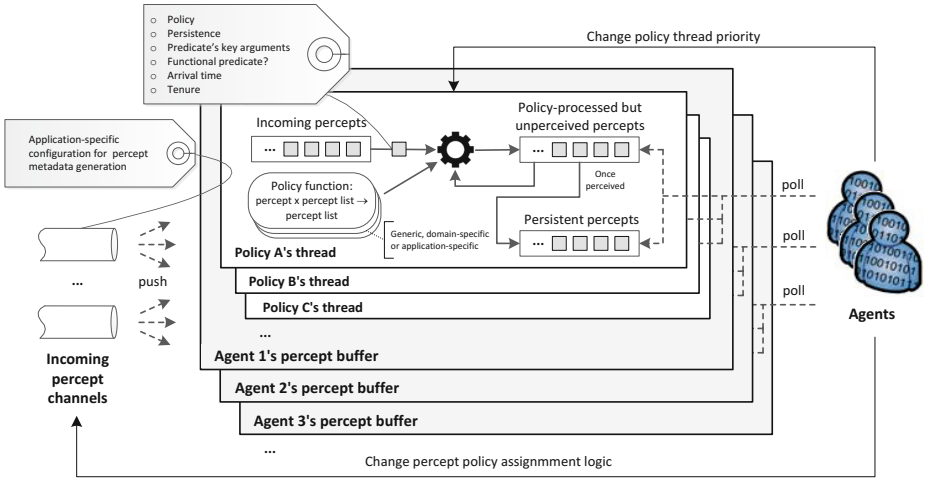

Fig. 1. The architecture and interfaces of a percept buffer

and whether or not the percept's predicate is *functional*, i.e. whether it can only have a single value at any time for given arguments at the key argument indices. Percepts are also specified as being *transient* or *persistent*. Transient percepts are only stored until the agent's next perception, whereas persistent percepts are treated as part of the environment's state, and are also perceived by the agent in subsequent perceptions (unless replaced by newer percepts, or they expire as specified by the percept's *arrival time* and *tenure*). Note that the aim of the percept buffer is not to act as the agent's memory in general. However, we see its role as providing an agent environment that encapsulates the external sources of percepts. We therefore allow an agent developer the option of using the buffer to store persistent state that may not be made available repeatedly by the external system.

The percept metadata, and the implementations of the policies used (conforming to a simple interface—see Sect. 6), provide the domain- and application-specific information used by the percept buffers. Therefore, configuring our approach for a specific application involves providing a mechanism for the channels to add the required metadata, e.g. application-specific rules. More detail on our metadata scheme is given in Sect. 5.

The architecture allows agents to dynamically control their perception by changing how channels assign policies to percepts (based on their functor and arity), and the priorities of the threads that execute policies. The mechanisms for providing this functionality will depend on the agent platform used. Our implementation, using Jason [3], provides agent "internal actions" for this purpose.

Each agent has its own percept buffer, which has percepts, along with their metadata, pushed to it from the channels. A single percept may be delivered at a time, e.g. when a stream of data is being consumed by a channel, or a set of percepts may be delivered together, with the intention that these represent a complete state update for the agent. In the latter case, we assume that there is

a single channel or that the channels have been designed so that the buffer does not need to synchronise state updates from different channels. We also assume that all percepts in the state update are to be processed by the same policy[3], or that it does not matter if a single state update results in different policies' outputs being perceived by the agent at different times[4].

For each percept-processing policy in use, a percept buffer maintains a (thread-safe) queue of incoming percept sets that have been pushed on the queue by the channels. Each percept set on the queue is either a singleton set (in the case of percept streaming) or represents a state update. In addition, for each policy, there is a list of processed but unperceived percepts and a list of previously perceived but persistent percepts. These contain the buffered percepts that are waiting to be delivered to the agent when it next perceives the environment. The latter list contains percepts that should be repeatedly delivered to the agent, according to the percept metadata. The percept buffer creates a thread for each policy that repeatedly takes percept sets from the incoming queue and combines them with the buffered percepts to produce an updated list of buffered percepts.

When the agent perceives, it consumes the percepts in the unperceived percept list. It also receives percepts from the persistent percept list. At this time, the persistent percept set is updated with the newly perceived percepts that are annotated as being persistent. This may involve some instances of functional percepts being replaced with new ones. As there is a separate perceived persistent percept set for each policy, we require that functional persistent percepts with a given functor and arity are always associated with the same policy; otherwise the updating of persistent percepts cannot be guaranteed to be done correctly.

5 Percept Metadata

The following metadata scheme is used by channels when annotating percepts before delivering them to the agents' percept buffers. In this way, domain- and application-specific knowledge can be provided on how percepts should be treated.

Policy This metadata element specifies the name of the policy that should be used to combine a new percept with any 'matching' ones that have been processed by the policy but not yet perceived.

Persistent This element can be true or false, depending on whether the percept should be stored in the percept buffer persistently and repeatedly perceived by the agent until it is replaced by newer information or it expires.

KeyArgs As described above, the key arguments for a percept are those that comprise a compound key. The value of this optional element is a list of argument indices. This defines which processed but unperceived percepts match a

[3] It is possible for a single policy to process percepts with different functors.

[4] Our current implementation adds an additional assumption: that all percepts in an incoming percept set have the same persistence (transient or persistent), but this is simpler to remove than the other assumptions.

new one: percepts match if they have the same functor, arity, and values at the key argument indices.

FuncPred This has value `true` if the percept is an instance of a predicate that is functional, i.e. only one instance of the predicate can exist for any specific values of the key arguments. Subsequent percepts with the same key arguments must replace older ones. This is only used when updating the perceived persistent percepts. This is because policies have the responsibility of deciding how to resolve the co-existence of new and old matching unperceived percepts—the developer may wish the agent to receive all percepts that have arrived since the last perception, or an aggregation or summary of them.

ArrivalTime This records the time at which the percept arrived.

Tenure This optionally specifies an interval after which the percept is no longer useful and should be deleted even if not perceived. This is most useful for persistent percepts.

```
public interface Policy {
  public List<WrappedPercept> applyPolicy(
    WrappedPercept percept,
    List<WrappedPercept> queuedPercepts);

  public List<WrappedPercept> eventToStatePercepts(WrappedPercept p);

  public WrappedPercept transformPerceptAfterPerception(WrappedPercept p);
}
```

Fig. 2. The policy interface

6 Defining and Applying Policies

A policy is defined by a class that implements the interface shown in Java in Fig. 2[5]. The key method is `applyPolicy`. This is called for each percept in the new percept set in turn. The first argument, of class `WrappedPercept`, represents a newly received percept, wrapped by another object recording its metadata. The second argument, `queuedPercepts`, should be a list of the percepts that have been previously output by this method, are not yet perceived by the agent, and which match the new percept based on the functor, arity and KeyArgs metadatum. As new percepts arrive, the `applyPolicy` method will be repeatedly called to combine newly arrived percepts with those queued for perception by the agent. For some policies this will result in reducing the number of percepts received by the agent on each perception. By providing application-specific policy classes, the developer can customise how this is done. Some example policies were outlined in Sect. 3.

The other two methods in the policy interface are optional (they can just return a null value) and are discussed in Sections 7 and 8.

Pseudocode for the `run` method of a policy thread is shown in Algorithm 1. The key line of the algorithm is line 19, which obtains the application-specific

[5] A separate policy factory class is used to associate names with policy classes.

Algorithm 1. The policy thread's algorithm

```
    Data: policyName: Name of policy handled by this thread
          newPerceptQueue: Blocking queue of percept sets
          unperceivedPercepts: Concurrent map from policy names to percept list
          partitions
 1  forever do
 2      newPercepts ← newPerceptQueue.take()
 3      oldPercepts ← unperceivedPercepts.get(policyName)
 4      if oldPercepts = null then
 5          oldPercepts ← empty percept list partition
 6      end
 7      applyPolicyToAllPercepts(policyName, newPercepts, oldPercepts)
 8      oldValue ← unperceivedPercepts.replace(policyName, oldPercepts)
 9      if oldValue = null and oldPercepts ≠ null then
            // The agent has concurrently consumed old percepts
10          oldPercepts ← empty percept list partition
11          applyPolicyToAllPercepts(newPercepts, oldPercepts)
12          unperceivedPercepts.put(policyName, oldPercepts)
13      end
14  end

15  Procedure applyPolicyToAllPercepts(policyName, newPercepts, oldPercepts)
16  foreach p ∈ newPercepts do
17      key ← partitionKey(p)
18      matchingPercepts ← oldPercepts[key]
19      processedPercepts ← getPolicyObject(policyName).
                                applyPolicy(p, matchingPercepts)
20      oldPercepts[key] ← processedPercepts
21  end
```

policy object for the given policy name and calls the `applyPolicy` method. For brevity, in the algorithm we write "percept" to mean wrapped percept (a percept with its metadata). The percepts processed by the policy but not yet perceived, as well as the perceived persistent percepts, are represented as "percept list partitions". This data structure stores a list of percepts as a set of sublists. Each sublist contains the percepts with a given *partition key*: a triple combining a functor, arity and specific tuple of values for the key arguments of the predicate with that functor and arity, e.g. $\langle sensor\_reading, 2, \langle sensor72 \rangle \rangle$. This is a special case of a map, and we write $p[k]$ for the sublist of percept list partition p with partition key k.

The algorithm runs an infinite loop that takes each (possibly singleton) set of new percepts from the new percepts queue and processes it. Line 2 retrieves a set of new percepts and line 3 looks up the percepts that have been previously processed by this thread but not yet perceived. Lines 4–6 create a new percept list

partition if there are no previously processed but unperceived percepts. Line 7 calls a procedure (lines 15–21) that, for each new percept, looks up the matching percepts in the percept list partition, gets the policy object and applies it, and then updates the percept list partition with the results. The main algorithm (line 8) then checks whether the list of previously processed percepts for this policy, stored in `unperceivedPercepts` with the policy name as a key, has been consumed by the agent since the policy thread last retrieved it. The agent signals that this has occurred by removing the concurrent map entry for that key. In this case, the policy thread applies the policy to all new percepts starting with an empty percept partition list as the list of old percepts (lines 10–12). These policy applications cannot be skipped in case the policy is designed to change the structure of the incoming percepts, as in the "keep latest with history" policy described in Sect. 3.

The policy thread runs concurrently with the channels, which add new percept sets to `newPerceptQueue`, and the agent, which consumes the percepts stored in `unperceivedPercepts` for each policy. Therefore, the algorithm must be defined in terms of thread-safe data structures to ensure correct behaviour. In particular, we have chosen the `BlockingQueue` and `ConcurrentMap` data structures provided by Java for the implementations of `newPerceptQueue` and `unperceivedPercepts`, respectively. The `take` method (line 2) is used to retrieve a set of new percepts, and if the queue is empty, this method will block until a channel adds new percepts to the queue. Line 8 calls the `replace` operation on a concurrent map. This is an atomic operation that replaces the value for a given key in the map, and returns the previous value, or null if there was no previous value.

7 Agent Perception

Algorithm 2 presents the procedure run when the agent initiates a perception. For each policy in use, the percepts output by the policy but not yet perceived are retrieved, along with the persistent percepts, and added to the result set to be returned to the agent. In line 5, the `remove` method is called on the concurrent map `unperceivedPercepts`. This is an atomic operation to remove the map's value for the given key (the policy name in this case) and return the value retrieved, or `null` if there was no value. Removing the value signals to the policy thread that the percepts have been (or are in the process of being) perceived.

Lines 10–18 handle persistent percepts. In line 10 the policy's `eventToStatePercepts` method is invoked on the percept. This allows a single percept from a channel (e.g. an update for some element of the state) to be translated to a set of percepts representing the updated (persistent) state information. This is described further in Sect. 8. If there is a non-null result from this call, the original policy-processed but unperceived percept p is treated as a representation of a transient event and added to the set of percepts to be returned to the agent. The transformed 'state percept' is then passed to procedure `updatePersistentPercepts` to update the persistent state. If

Algorithm 2. Handling an agent request for percepts

```
    Data: unperceivedPercepts, persistentPercepts: Concurrent maps from policy
          names to percept list partitions
 1  Function perceive(): Set of percepts
 2  result ← empty list
 3  currTime ← current time
 4  foreach policyName in keys of unperceivedPercepts do
 5      newPerceptsPartition ← unperceivedPercepts.remove(policyName)
 6      persPerceptsPartition ← persistentPercepts.get(policyName)
 7      if newPerceptsPartition ≠ null then
 8          foreach p ∈ newPerceptsPartition do
 9              if p.isPersistent() then
10                  statePercepts ←
                      getPolicyObject(policyName).eventToStatePercepts(p)
11                  if statePercepts ≠ null then
                        // There are separate event and state representations
                        // The event percept goes directly to the agent
12                      addUnwrappedPerceptIfNotExpired(p, result, currTime)
13                      foreach statePercept ∈ statePercepts do
14                          updatePersistentPercepts(statePercept, currTime,
                              persPerceptsPartition)
15                      end
16                  else
17                      updatePersistentPercepts(p, currTime,
                          persPerceptsPartition)
18                  end
19              else
20                  addUnwrappedPerceptIfNotExpired(p, result, currTime)
21              end
22          end
23      end
24      foreach p ∈ persPerceptsPartition do
25          afterPerceptionPercept ← getPolicyObject(policyName).
26                                       transformPerceptAfterPerception(p)
27          if afterPerceptionPercept ≠ null then
28              Remove p from persPerceptsPartition
29              Add afterPerceptionPercept to persPerceptsPartition
30          end
31          addUnwrappedPerceptIfNotExpired(p, result, currTime)
32      end
33      persistentPercepts.put(policyName, persPerceptsPartition)
34  end
35  return result
```

`eventToStatePercepts` returned `null`, the unmodified percept is passed to that procedure.

The algorithm for `updatePersistentPercepts` is not shown due to lack of space. This uses the percept's partition key (its functor, arity and key argument values) to obtain the sublist of `persPerceptsPartition` that matches the percept. If the percept's predicate is functional (according to the percept metadata), the matching percepts are removed from that sublist. If not, any expired percepts are removed from the sublist (using their `ArrivalTime` and `tenure` metadata, if present, and `currTime`). In either case, the (still wrapped) percept is added to the sublist if it has not expired.

Transient percepts are handled in line 20. They are added to the result set if not already expired.

Finally, in lines 24–32 all persistent percepts are added to the result set. There is one wrinkle here. The policy may have added extra information to the percept, as in the "keep latest with history" policy described in Sect. 3. The policy method `transformPerceptAfterPerception` gives developers the option to remove this extra information from persistent percepts if it should only be perceived once.

8 Events and States

As discussed above, Algorithm 2 calls two optional policy methods: `eventToStatePercepts` and `transformPerceptAfterPerception`. The role played by these methods has been explained above. In this section we briefly explain the motivation for these methods.

Plans in a BDI agent program can be triggered by the addition of new beliefs to the agent's belief base. The belief base can also be queried from within the context conditions or bodies of its plans. These illustrate two different uses of percepts within a BDI program: (i) to react to new information by triggering a plan, and (ii) to look up previously received information in the course of instantiating or executing a plan. We believe that in many agent programs this distinction corresponds to the difference between using percepts to encode (i) events, and (ii) state information. However, it is also the case that some percepts can represent both an event and state information. In particular, a percept may encode a change of state, and may be used in the agents' plans both to trigger a plan and for looking up the current state at a later time. Our design for percept management policies aims to support developers in achieving separation of concerns when handling event and state information in their agent plans. Specifically, the policy method `eventToStatePercepts`, shown in Fig. 2 and used in Algorithm 2, will be applied to a policy-processed persistent percept p, just before the agent perceives it. The method can return `null` if this functionality is not required. Otherwise, the result is a list of percepts, which encode the information in the original percept in a different way for storage in the persistent percept list. The original percept p is treated as transient and sent to the agent once only.

For example, a percept $approved(ag, doc, stg)$ received from a channel may indicate that agent ag has approved document doc to move to stage stg of a publishing workflow. The policy method `eventToStatePercepts` can be used to generate a persistent record of the state of the document, e.g. $doc\_state(doc, stg)$.

9 Case Studies

We have implemented a prototype percept buffer by extending our open source *camel-agent* software [5]. This provides a connection between the Jason BDI agent platform [3] and the Apache Camel message routing and mediation engine [9]. We use Camel message-processing *routes* as our channels. These routes receive information from external systems using Camel's wide range of endpoints for various networking technologies and protocols. The resulting Camel messages are transformed and filtered as required, using one of Camel's domain-specific languages. Percept metadata is added in the form of message headers, and the messages are then delivered to camel-agent's *agent percept endpoints*. These use endpoint configuration information or Camel message headers to identify the recipient agents(s), and the messages are then delivered to these agents' percept buffers.

We also provide Java implementations for Jason internal actions to dynamically control the processing of percepts within the percept buffer by altering the logic used by channels to assign policies to percepts, and by changing the priorities of policy threads.

To demonstrate and evaluate the use of percept buffers, we developed policies to handle three different sources of streaming data: two demonstration data streams on the web and a live stream of events from a Minecraft server.

9.1 Demo Data Streams

We first evaluated the utility of our approach by configuring channels to consume data from two data streams streamed live over the web by PubNub, Inc.[6]: the Game State Sync stream and the Sensor Network stream. These provide simulated data streams described as (respectively) "updated state information of clients in a sample online multiplayer role-playing game" and "sensor information from artificial sensors". For each of these data streams we used the PubNub Java client library to create a channel that subscribes to the stream and sends the data received (translated to Jason syntax), along with the required percept metadata, to the queue of incoming percepts for the single agent used in this scenario. For the Game State Sync stream, the channel produces a single percept for each data item on the stream. For the Sensor Network stream, a single data item is converted to four percepts recording different aspects of the sensor reading.

The formats of the data items in the two streams are shown below, after translation to Jason literals.

Game State Sync:

```
action(PlayerId, CoordX, CoordY,
       ActionName, ActionType, ActionValue)
```

[6] http://www.pubnub.com/developers/demos/

Sensor Network:

radiation(SensorUUID, Radiation)
humidity(SensorUUID, Humidity)
photosensor(SensorUUID, LightLevel)
temperature(SensorUUID, Temperature)

Consuming the Game State Stream. As the messages received from the Game State Sync stream represent events, we configured the channel connected to this stream to mark all action percepts as transient (and so the FuncPred metadata element is not relevant). As the stream uses (seemingly) randomly generated three digit numbers as identifiers in action percepts, the chance of two or more matching agent IDs occurring between consective agent perceptions is very low, so we did not specify any key arguments for the action predicate. This means that all action percepts match each other. A simple, but non-trivial, Jason agent program was used to handle the percepts received[7]. We investigated the effect of three different policies for handling these percepts. Our purpose here is not to analyse or criticise the operation of any specific agent platform (and Jason in particular), but to illustrate the problems that arise when handling streams of percepts.

First, using the policy "keep latest percept" as a baseline case confirmed (not surprisingly) that buffering is needed when percepts are being produced and consumed asynchronously. This policy stores no more than one percept between consecutive agent perceptions. During a ten minute run, 5625 messages were received from the Game State channel (9.4 messages per second). Although Jason's perception rate was significantly higher (an average of 60.2 per second), 404 percepts were lost (7.2%) due to the lack of buffering. In addition, although 5221 action percepts were delivered to the agent, there were only 5216 plan invocations[8]. The missing plan invocations were not just delayed slightly—after an additional minute the count was the same.

In another ten minute run using the default policy (to queue all percepts until they are perceived), 5369 messages (all distinct) were received on the channel and these were all delivered to the agent. However, there were only 5260 plan invocations, suggesting that Jason was unable to cope with this load. For this, and the previous policy, similar results were observed in a previous run (which used an older version of Jason).

A final run was performed using the "keep latest with history" policy. For each set of matching percepts (as determined by the KeyArgs metadata element), this policy retains only the latest percept in the unperceived percepts data structure, but stores a list of older matching percepts within an additional argument (or by using some other method provided by the agent platform for adding information

[7] The plan handling action percepts updates a belief counting plan invocations, checks that the player ID is not in a given five-element list (chosen to never match any player IDs), and calls a subgoal that is handled by a plan with the trivial body 'true'. Ten other trivial plans handle belief additions that never occur.

[8] All percepts were distinct, and therefore were genuinely new beliefs.

to percepts—we used a Jason *annotation*). The result is fewer percepts for the agent plans to handle, and the programmer can choose under what conditions the history of older recent percepts should be examined.

When using this policy, 5597 messages were received on the channel during a 10 minute run. Fewer percepts, 5111, were delivered to the agent when using this policy, but there were still two plan invocations missing. Similar results were observed in a second run, when three plan invocations were missing. In this case the percept buffer and choice of policy have not completely solved the problem of missing plan invocations. Jason has a configuration option to set the number of BDI reasoning cycles that are performed beween two consecutive perceptions. Setting this to 2 allowed the "keep latest with history" to further amalgamate percepts between perceptions, and 5416 percepts from the channel were amalgamated into 1096 percepts delivered to the agent. All these led to plan invocations, Two more runs produced similar results.

These results show that setting appropriate policies in a percept buffer can significantly reduce the number of percepts that a BDI plan must handle. However, it may also be necessary to control the rate of agent perception to allow the buffer time to amalgamate or summarise percepts over a longer period of time.

Consuming the Sensor Network Stream. In this section we use the sensor network stream to demonstrate how the percept buffer gives developers the flexibility to customise the delivery of percepts to the agent.

First, we consider default percept metadata settings that label all percepts as being transient and to be queued until perceived (the default policy). As the first argument of each of the sensor reading predicates is the sensor identifier, we declare this to be the key argument. However, for this setting to be useful we had to customise the channel to replace the sensor identifier with a random number from 0 to 19—the stream unrealistically uses random IDs that never appear to reoccur. For the purposes of our discussion, we assume that the agent is only interested in monitoring radiation settings, and the agent has a plan to count these percepts, as well as two more plans that handle percepts related to reporting (and which only consist of a `println` action). With these settings, during a ten minute run, 22508 percepts were delivered to the percept buffer. A quarter of these (the 5627 radiation percepts) should have triggered plan invocations, but only 5307 plan invocations were counted.

We next considered the combined use of two policies. The radiation percepts were handled by a policy that, for given key argument values, keeps a single percept with an added timestamp in the unperceived percepts list (using a Jason annotation). Also, when a new percept arrives and a matching unperceived percept is present, the policy keeps whichever of the two has the maximum radiation reading. This assumes that the agent is monitoring for peak readings and should not miss any. The other percepts were sent to a policy that ignores them by simply removing them from the incoming queue. With this combination of policies, 5622 messages on the channel resulted in 5613 percepts delivered to

the agent, all of which resulted in plan invocations. This demonstrates that for this application, filtering out the unwanted percepts by using the "ignore" policy achieved a better outcome than delivering them and letting the agent code ignore them. The use of a "keep maximum" policy had little effect on reducing percept numbers, but ensured the agent would not miss the most significant events.

The final policy we consider is one that converts events to state information using the `eventToStatePercepts` method. We note that the stream does not deliver information for all sensors at once—sensor readings arrive one at a time. We assume that the developer wishes to treat the received sensor readings as state information that can be queried in plan context conditions and bodies and not just as events that trigger plans. Therefore we configured the channel to label the radiation percepts as persistent. However, the readings are time-dependent and lose their validity over time, so we set a 10 second tenure period for percepts. We specify that the first argument of the radiation predicate has no key arguments. This allows a policy to collect all unperceived percepts with this predicate into a list, wrapped in a radiation_list percept. On agent perception, this is sent as a one-off percept to the agent, while a set of persistent percepts are produced by the `eventToStatePercepts` method. The persistent percepts use a functor (radiation_state) that is different from the original percepts. This predicate is specified as functional with its first argument being the key argument, so that the persistent percepts are appropriately maintained over time. With this configuration, over a ten minute run, 5748 messages were collected into 5491 radiation_list percepts that were delivered to the agent, all of which resulted in plan invocations (although, it should be noted that the plan is very simple: it just updates a count belief). In addition, the persistent percepts accounted for another 500087 percepts. These included repeated percept deliveries, which would cause no "new percept" events to be output from Jason's belief update function, but also prevented Jason from removing these percepts from the agent's belief base.

9.2 Sensing Data from Minecraft

An additional case study involved connecting the percept buffer to a channel linked (via a web socket) to a mineflayer[9] JavaScript bot for Minecraft. Minecraft[10] is a single or multiplayer game in which players mine the environment for materials, construct buildings, and (in "survival mode") fight monsters. We investigated the impact of the percept buffer on the speed of an agent performing a specific sensory task over a stream of events from Minecraft. The events represented the position of the bot and the movements of various creatures in the simulated world, and the task was to detect ten distinct squid and then ten distinct bats within a certain range. This task can be achieved using a simple Jason program comprising two short plans, but as 100–200 events arrive per second, we endeavoured to provide a policy to ease the task. Our policy treated

[9] https://github.com/andrewrk/mineflayer
[10] https://minecraft.net

the percepts as transient, and ignored percepts from outside the specified range as well as percepts related to creatures other than the target species (initially squid). It also kept only the latest unperceived percept for a given individual creature. We connected two agents to the same Minecraft event stream. One used our special policy, while the other used the null policy (buffering only). The channel was configured to treat percepts recording the bot's own position as persistent for both agents. The Jason plan for the null policy agent performed range checking as well as counting and tracking which of the target creatures had already been seen (using their identifiers). The plan for the agent with the special policy did not need to perform range checking, and received a smaller number of percepts. Once the first part of the task was completed (counting 10 distinct squid), the plan used an internal action to request the channel to change the policy used for its percepts so that only bat percepts were delivered to it. This demonstrates the ability to change policies dynamically to change an agent's focus of attention.

Unfortunately the task performance times for the two agents were almost identical to within a few milliseconds for each of eight runs. This is probably due to the task needing only simple plans that can do all necessary percept filtering using plan "context conditions", for which Jason is (presumably) well optimised. However, this case study demonstrates that the use of the percept buffer allowed the agent code to be simplified and did not add any overhead for the performance of the task, even though the performance was not improved.

10 Conclusion

This paper has presented a design for an agent percept buffer to simplify the handling of percepts from external systems—especially high frequency streams. Rather than relying on programmers to build a custom agent environment encapsulating external sources of percepts, a percept buffer provides a generic solution that can be customised for a given application. This is done by (a) configuring the channels that deliver percepts to the buffer to attach domain-specific information about those percepts, and (b) providing appropriate application-specific policies. This work provides the first platform-independent and detailed proposal for addressing a problem that is often faced, but which must currently be tackled in an ad hoc application-specific manner.

We defined the architecture and algorithms for processing percepts in the percept buffer and for responding to perception requests from agents. We also defined a percept metadata scheme used for providing the buffer with domain-specific information about the percepts. Three case studies were presented to illustrate the flexibility offered by our approach for handling percept streams, and to evaluate its benefits.

Future work includes extending the metadata scheme to allow the absence of certain percepts in a stream to be considered significant, based on some form of local closed world reasoning. Further experimentation with larger and more realistic applications is also needed.

References

1. Behrens, T.M., Hindriks, K.V., Dix, J.: Towards an environment interface standard for agent platforms. Annals of Mathematics and Artificial Intelligence 61, 261–295 (2011)
2. Bogdanovych, A., Rodriguez-Aguilar, J.A., Simoff, S., Cohen, A.: Authentic interactive reenactment of cultural heritage with 3D virtual worlds and artificial intelligence. Applied Artificial Intelligence 24(6), 617–647 (2010)
3. Bordini, R.H., Hubner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. Wiley (2007)
4. Bratman, M.: Intention, plans, and practical reason. Harvard University Press (1987)
5. Cranefield, S., Ranathunga, S.: Embedding agents in business processes using enterprise integration patterns. In: Winikoff, M. (ed.) EMAS 2013. LNCS, vol. 8245, pp. 97–116. Springer, Heidelberg (2013)
6. Dennis, L.A., Fisher, M., Lincoln, N.K., Lisitsa, A., Veres, S.M.: Declarative abstractions for agent based hybrid control systems. In: Omicini, A., Sardina, S., Vasconcelos, W. (eds.) DALT 2010. LNCS, vol. 6619, pp. 96–111. Springer, Heidelberg (2011)
7. Gemrot, J., Brom, C., Plch, T.: A periphery of Pogamut: From bots to agents and back again. In: Dignum, F. (ed.) Agents for Games and Simulations II. LNCS, vol. 6525, pp. 19–37. Springer, Heidelberg (2011)
8. Hindriks, K.V., van Riemsdijk, B., Behrens, T., Korstanje, R., Kraayenbrink, N., Pasman, W., de Rijk, L.: UnREAL Goal bots: Conceptual design of a reusable interface. In: Dignum, F. (ed.) Agents for Games and Simulations II. LNCS, vol. 6525, pp. 1–18. Springer, Heidelberg (2011)
9. Ibsen, C., Anstey, J.: Camel in Action. Manning Publications Co. (2010)
10. Jason-users: Update rate of Jason. Thread on Jason-users mailing list (2014), http://sourceforge.net/p/jason/mailman/message/29859084/
11. van Oijen, J., Dignum, F.: A perception framework for intelligent characters in serious games. In: Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems, IFAAMAS, pp. 1249–1250 (2011)
12. Oijen, J., Poutré, H., Dignum, F.: Agent perception within CIGA: Performance optimizations and analysis. In: Müller, J.P., Cossentino, M. (eds.) AOSE 2012. LNCS, vol. 7852, pp. 99–117. Springer, Heidelberg (2013)
13. Ranathunga, S., Cranefield, S., Purvis, M.: Identifying events taking place in Second Life virtual environments. Applied Artificial Intelligence 26(1-2), 137–181 (2012)
14. So, R., Sonenberg, L.: The roles of active perception in intelligent agent systems. In: Lukose, D., Shi, Z. (eds.) PRIMA 2005. LNCS, vol. 4078, pp. 139–152. Springer, Heidelberg (2009)
15. Wei, C., Hindriks, K.V.: An agent-based cognitive robot architecture. In: Dastani, M., Hübner, J.F., Logan, B. (eds.) ProMAS 2012. LNCS, vol. 7837, pp. 54–71. Springer, Heidelberg (2013)
16. Ziafati, P., Dastani, M., Meyer, J.J., van der Torre, L.: Event-processing in autonomous robot programming. In: Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems, IFAAMAS, pp. 95–102 (2013)